Operating Systems

Scheduling
Based on Ch. 5 of
OS Concepts by SGG

Scheduling

- If we have more processes than cores we have to decide which ones run and which ones wait
- In other words we have to do scheduling
- We have to make sure that
  - The CPU and the rest of the system are used efficiently
  - The we do not have starvation
  - The users are satisfied
  - Priorities are respected
  - etc.

The life of a Process

- From a scheduler point of view, a process alternates between two states
  - CPU burst
  - I/O burst
- The CPU scheduler is sometimes referred to as short-term scheduler
- There are two kinds of schedulers
  - Preemptive
  - Non preemptive

The Life of a Scheduler

- The simplest would be to have a ready queue and any time it is activated it moves the running process to the end of some queue and moves the head of the ready queue to run on a CPU
- This would work but not very well
  - For example: if there are 10 I/O-bound process waiting for a cpu-hog

When Scheduling Takes Place

- A scheduler may be activated in one of four situations
  - 1. A process switches from running to waiting (blocks)
  - 2. A process is temporarily paused because of some interrupt
  - 3. An I/O (or other operation) completes
  - 4. When a process terminates
- If scheduling occurs in 1 and 4 only: non-preemtive
- If scheduling occurs in all four: preemptive

Preemptive Schedulers

- Preemptive schedulers are slightly more complicated
- The difficulty is that we may have race conditions and need synchronization
- The kernel itself might not be preemptive even on preemptive systems.
  - Simple to use and verify
  - Unsuitable for real time use
- Most modern kernels are preemtable

Context Switching

- Context switching involves
  - Switching to monitor mode
  - Saving the state of the current process in the CPU
  - Loading the state of another process
  - Switching to user mode and jumping to the saved PC
- Context switching is rather expensive
  - Page table installation
  - Clearing of cache
  - Etc.
- The program that does the switching is called the dispatcher
- We can monitor the dispatcher activity on Linux with vmstat or through *proc/*<PID>/status

Scheduling Criteria

- CPU utilization
- Throughput
- Turnaround time
- Waiting time
- Response time

Basic Scheduling Algorithms

- First Come First Served
  - Simple and fair, but has large avg waiting time
- Shortest Job First
  - Is the best in terms of waiting time
  - Has issues with starvation
  - We need to guess the future!
    - There are methods to guess the duration of the next CPU burst
- Shortest Remaining Time First
  - The preemptive sister of SJF
- Priority Scheduling
- Round Robin
  - The most suitable for interactive systems
  - Has long waiting times

SJF

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

Round Robin

- RR has long waiting times
- Every process runs for at most one time slice or time quantum
  - If not done before the end of the slice it is preempted
- With a huge time slice RR becomes FCFS
- With a tiny slice we spend most time in context switching

Multilevel Queue Scheduling

- The system has multiple queues of increasing priority
- When a high priority queue is empty, processes from lower priority queues can run
- They are preemptive, in general.
- Has problem with starvation
  - One solution is aging
- It is also inflexible
  - What happens if the priority depends on the predicted duration of the CPU burst and the prediction is wrong.

Multilevel Feedback Queues

- Initially all processes enter the high priority queue. It runs RR with a short time slice
- If they use up their slice in one go the are demoted to the next lower queue that has RR with longer slice
- If they use that too, they are demoted further
- The lowest priority queues have FCFS

Thread Scheduling

- In user level threads the kernel (and thus the scheduler) knows nothing about the threads
  - All the scheduling is done within the time allotted to the mother-process
  - We call it *process contention scope*
- In kernel level threads all the threads compete for the CPUs, even against threads that belong to a different process
  - We call this *system contention scope*.

Pthread Scheduling

- Pthreads can work in both process and system contention scope (in theory).
- Most systems only implement one or the other
- The API provides two functions for this
  - pthread_attr_setscope
  - pthread_attr_getscope

Multi-Processor Scheduling

- There are two approaches
  - Asymmetric: one processor does the scheduling and other kernel jobs (no need to synchronize)
  - Symmetric: each processor is self scheduling. This is more complex as all processors need to access shared data
- Most systems do Symmetric Multi-Processing with either
  - A global ready queue
  - A per core ready queue to keep contention/synchronization low.

Affinity

- Processor affinity, ie what processes go to what cores (reduce cache invalidation, enforce I/O requirements, etc)
  - Hard affinity
  - Soft affinity
- Many computer systems rely on affinity mainly to improve cache performance (UMA systems), or memory speed (NUMA)
- Other computer systems rely on affinity to function properly (eg h/w connected to one core only)
- Heterogeneous Multiprocessing systems rely on affinity to conserve power (big.LITTLE ARM processors like Exynos 5)

Load Balancing

- With multiple processes/cores we may need to do load balancing
  - Interferes with affinity
  - Needed on systems where every processor has its own queue
- There are two schemes (often mixed)
  - Push balancing (a separate process keeps an eye on load balance)
  - Pull balancing (idle cores “pulling” load from other cores)

Multi-core and multi-threading

- In a modern system memory is much slower than the CPU (many dozens of cycles to read from the memory
- This results in long memory stalls
- If a processor could run more than one thread at a time it can switch to another thread during a stall (computer architecture jargon: “thread” can mean both thread and process)
- A two way multithreaded core appears as two cores to the OS.

Multi-core and multi-threading

- There are two general multithreading schemes
  - Coarse grained (only for long stalls, needs to flush the pipeline)
  - Fine grained (the pipeline contains a mix of instructions from both threads, thread switching at almost every cycle)
- Scheduling becomes more complex: two threads on the same core run slower than on two cores.

Real time Scheduling

- Like most things in this lecture, there are two kinds
  - Soft real time systems (guarantee attention within certain time limits)
  - Hard real time systems (guarantee completion within certain time limits)

Minimize latency

- Latency is the time between an event occurring and being serviced
- Two types of latency
  - Interrupt latency: the time after the interrupt arrives where the processor finishes up what is doing and attends the interrupt
    - Disabling interrupts increases latency
    - Large number of registers to save increases it too.
  - Dispatch latency: the time to do context switch, free resources
    - preemptive kernels can help bound this latency.

Priority and Preemption

- When an event occurs that needs attention, we need both a priority scheduler and preemption
  - Even then we can guarantee attention, not completion.
- Hard real time systems must have
  - Periodic processes with constant period
  - Well defined processing time
  - Well defined deadline

Hard Real Time Scheduling

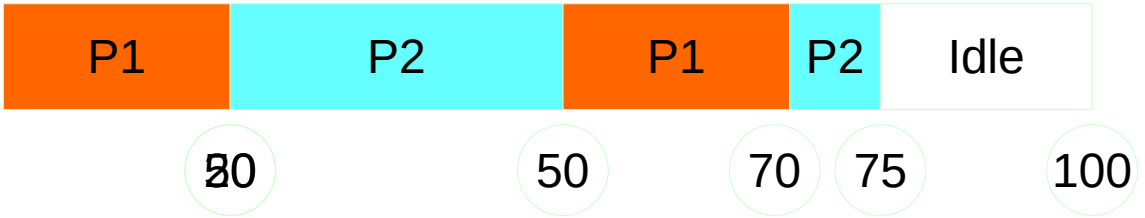
- All processes are considered periodic with period *p*.
  - This means they have a rate 1/*p*.
- They have a deadline *d*.
  - This means that the have to finish within *d* time from the start of the period.
- They have an execution time *t*.
  - It should be less than the deadline *d*.

Rate monotonic scheduling

- Priority is inverse of period (shorter period, higher priority)
- Every process has the same max processing time in every period
- Deadline is the start of next period
- The total CPU utilization must remain less than 100%
  - But not much less!
- It is the optimal among fixed priority systems.

Rate Monotonic Scheduling

- An example that works fine, but would not work with the priorities the other way around.



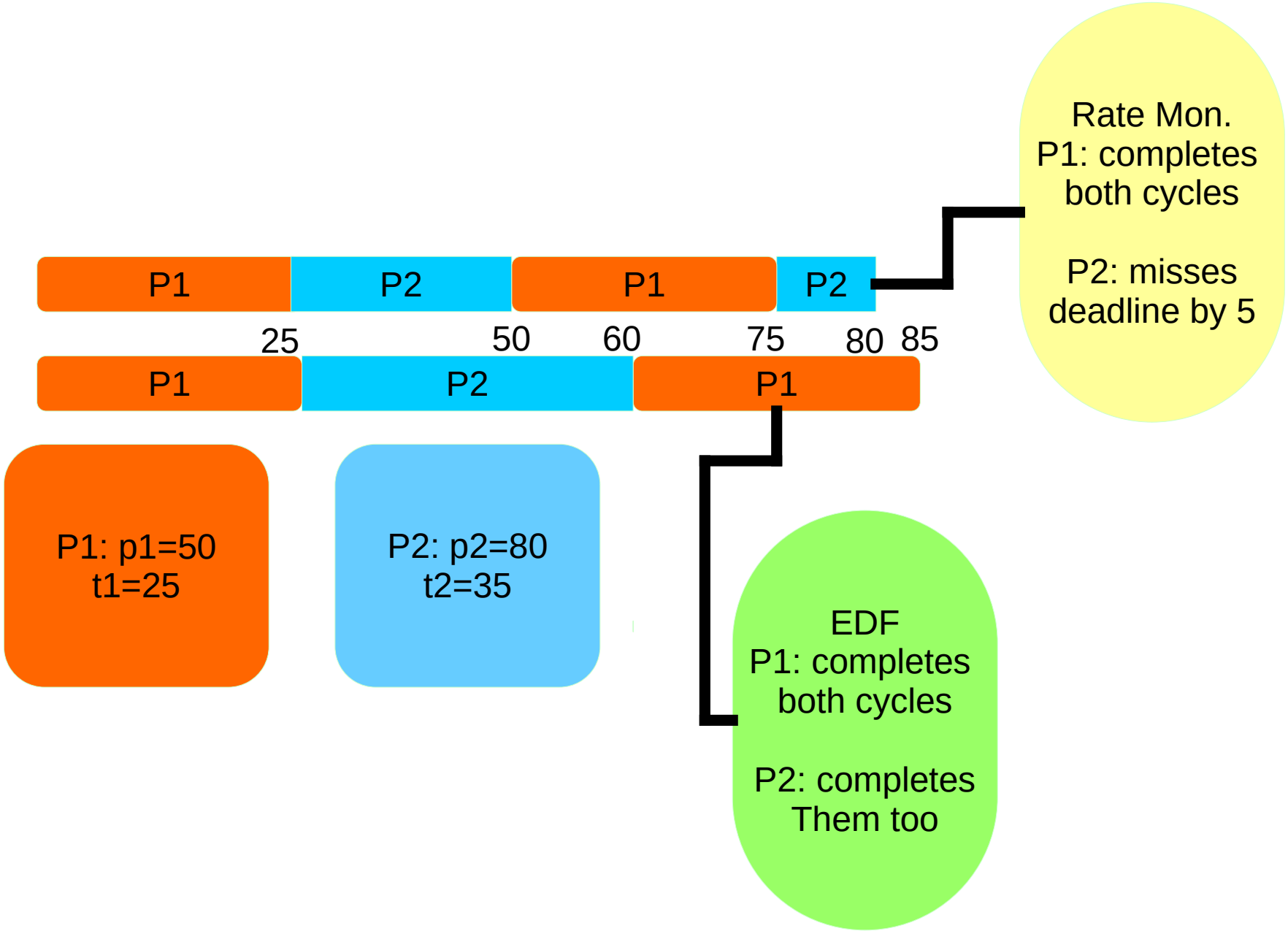
P1: p1=50
t1=20

P2: p2=100
t2=35

Rate-Monotonic Scheduling

- Has many nice properties but does not always work
  - Even if there is a solution.
- Can be proven that the CPU utilization cannot be 100% if we need to schedule two or more processes
- With two processes can go down to 83%
  - So it is easy to find a set of processes that cannot be scheduled with this method

Rate-Monotonic Scheduling



Earliest Deadline first

- Much better than rate monotonic
- Can work as easily for periodic and non periodic processes
  - But makes it harder to decide in advance if the deadlines will be met.
- It is theoretically the best
  - If we ignore context switching etc.
  - Priorities change from cycle to cycle.

Linux Scheduling

- Early Linux schedulers were similar to Unix ones:
  - Nothing real time or multicore about them
- Then multicore machines arrived and O(1) scheduler
  - O(1) was great for SMP
  - Not great for interactive processes
- Then CFS came (Completely Fair Scheduler)

CFS

- For non-real time priority classes
  - Gives more time to higher priority processes based on a virtual run time
    - The more CPU it used recently the smaller proportion of the CPU they get
    - The higher the priority (low nice number) the more CPU they get
- For real time priority classes
  - Allows pthreads to chose between SCHED_FIFO and SCHED_RR
  - The priority is fixed (no virtual time or nice number)
  - Higher priority jobs run first.

Windows Scheduling

- Has priority based preemptive scheduler
- Priorities 1-15 are variable class priorities
- Priorities 15-31 are real time priorities
- Processes get a priority and this can change for variable class processes
  - If it exceeds a time quantum goes down
  - If it just got unblocked goes up
    - If it was waiting keyboard input goes up a lot.

Algorithm Evaluation

- Deterministic Modelling
  - Examine characteristic, realistic or theoretical scenarios
  - Helps identify weaknesses
- Queuing Models (statistical models)
  - Very intuitive for simple schedulers
  - Give answers in simple formulas
  - Hard to extend to very complex systems

Queuing models

- Apply to anything from mechanical telephone networks, meteorite impacts, bus schedules, disk crashes, photons falling on a telescope.
- For example:
  - Little’s Law (or formula)
  - The average queue length is the arrival rate times the average waiting time.

Little's Law

$$n = \lambda T_w$$

$$n = \frac{T_{totw}}{N_{ticks}}$$

$$\lambda = \frac{N_{jobs}}{N_{ticks}}$$

$$T_w = \frac{T_{totw}}{N_{jobs}}$$

Simulations

- Give us a great way to evaluate a system with relatively low cost.
- Could use synthetic data or trace tapes from real systems
  - In other words both statistical and deterministic models.
- They typically require many hours of computations.